SORTING ALGORITHMS

A Survey of Basic Sorting Algorithms

1. Introduction

Sorting, or in simple terms ordering of data, is one of the elemental tasks for computer science applications towards searching data as search done on an ordered array/list would take less time. Sorting returns numbers in a particular order (high to low or vice versa) while strings in alphabetical order. There are many advanced sorting algorithms, however, we will only consider and analyze the most basic ones. We are well familiar with Bubble sort and Insertion sort, and so now we will analyze four comparison-based algorithms, which are also familiar and three non-comparison based algorithms along with some new hybrid sorting algorithms implemented in the 20's.

2. Comparison-Based Algorithms

2.1. Selection Sort

An in-place algorithm that finds the minimum or the maximum element (as per the requirement) from the unordered array/list and exchanges it with the first element in the ordered array/list, then repeats the same for the second, third, fourth…nth minimum/maximum and swaps with second, third, forth…,nth element respectively. It is better than Bubble Sort but worse than Insertion sort.

```
SORT(S)
1. for i =0 to size(S)
2.         for j= i+1 to size(S)
3.                   if  S[i] > S[j]
4.                             Swap S[i] with S[j]
```

It takes n-1 swaps and $O(n^2)$ comparisons to sort a list. It is inefficient for large arrays due to worst-case performance of $O(n^2)$. This works best with problems which require no extra memory or with values repeated in the array.

2.2. Quick Sort

Quick sort, a divide and conquer algorithm, requires no additional memory space for sorting. It sorts the elements by first choosing a pivot element A[q] and then partitions the input array into two subarrays such that each element of the first subarray is less than or equal to the pivot and all elements in second are greater than the pivot. Then, it recursively sorts the two subarrays and combines them to get the entire sorted array.

```
QUICKSORT (Arr, l, h)
1 if l<h
2     then pivot ← PARTITION (Arr, l, h)
3          QSORT (Arr, l, pivot−1)
4          QSORT (Arr, pivot+1, h)
```

```
PARTITION(Arr[],low,high)
1 var ← Arr[high]
2 i ← low −1
3 for j ← low downto high−1 do
4          if Arr[j] ≤ var
5                    then i←i+1
6 exchange Arr[i] ↔ Arr[j]
7 exchange Arr[i+1]↔Arr[r]
8 return i+1
```

This is the fastest sorting algorithm and has *O(n log n)* average case performance. If the array is sorted, and the pivot is either the leftmost or the rightmost, or if the array has all equal elements, the algorithm is inefficient and we get worst case complexity of $O(n^2)$.

2.3. Merge Sort

Merge sort is a stable sort divide and conquer algorithm with *O(n log n)* as the performance in worst case. It recursively splits the input array into n subarrays until each has one element. It then combines each subarray to form a sorted array until we are left with one subarray in sorted order. It is a stable sort algorithm

This algorithm is not efficient in the terms of its memory requirement. It is not an in-place algorithm and stores the sub-array and the main array in a new array.

```
MERGE-SORT(list, start, end)
1: if end>1
2:      midpoint = (start + end) /2
3:      MERGESORT(list, start, midpoint)
4:      MERGESORT(list, midpoint + 1, end)
5:      MERGE(list, start, midpoint, end)
```

```
MERGE(list, start, mid, end)
1. n1 = start
2. n2 = mid+1 , k=0
3. let S[end-start+1] be an array
4. for i ← n1 downto end do
5.      if n1 >mid
6.              S[ k++] = list [n2++]
7.      else if n2 >end
8.              S[k++] = list [n1++]
9.      else if list(ni) < list (n2)
10.             S[k++] = list[n1++]
11.     else
12.             S[k++] = list (n2++)
13. for j ← 0 downto k
14.     list [start++] = S[n1]
```

2.4. Heap Sort

Heap Sort is an in-place algorithm based on the heap data structure. The input list is used to build a heap and the sorting is done by moving the minimum value element (ascending order) at the last position(n-1) of our ordered array. The heap property is restored every time an element is removed until the heap is empty. It is not a stable sort algorithm, i.e., preservation of the order for equal keys is not guaranteed.

```
HEAP-SORT(list,size)
1. for i ← (length(list))/2 -1 downto 0
2.      heapify (list,size)
3. for i ← size-1 downto 0
4.      exchange(list[0],list[i])
5.      heapify(list, i,0)
```

```
heapify(list, size, i)
1. max =i
2. l = left(i)
3. r = right(i)
4. if (l <size and list [l] > list[max])
5.      max=l
6. if (r< size and list [r] > list[max])
7.      max=r;
8. if max is not equal to i
9.      exchange list[i], list[max]
10.     heapify(list, size, max)
```

Heap sort has run time performance *O(n log n)* (for both worst and average). Heap sort is considered efficient when it comes to sorting large data sets but it is not advised to be used to sort linked list.

3. Non Comparison-Based Algorithms

3.1. Radix Sort

Radix sort works by sorting the given values by keys which are in the form of mostly in integers, binary, or alphabets (in case for strings). It sorts every digit of the input element, beginning with the least significant digit and moving to the most significant. It is a stable sort algorithm. The average run-time complexity is *O(d.n)*, d being the count of digits in an element. However, if the number of digits varies it takes *O(log n)*. Radix sort is not a good choice when it comes to memory storage space.

```
RADIXSORT(A)
1: for j = 1 To digit do
2:        sort A on digit j using another sorting algorithm
```

3.2. Counting Sort

Counting Sort, a stable sort algorithm, is used to sort keys between a given range. It assumes that the input array consists of integers between 0 and n, n being the array size. It counts the number of time each element is present in the input array and stores this count in a new array. The final array is computed using some mathematical calculations on the data in this new array.

```
COUNTING-SORT(list,C,k)
2:  let C[0...k] be a new array
3:  for i ← 0 to  k do
4:        C[i]  ← 0
6:  for j ←1 to length(list) do
7:        C[list[j]]   C[list[j]] + 1
9:  for i ← 1; k do
10:       C[i]  ← C[i] + C[i - 1]
12.  for j ←  length(list) downto 1 do
13.  B[C[A[j]]]  ← A[j]
14.  C[A[j]] ←  C[A[J]] - 1
```

The worst case and average case time complexity for this sorting algorithm is *O(n+k)*. For maximum efficiency k must be smaller than n.

3.3. Bucket Sort

This is a distribution sorting algorithm. It splits the input array into subarrays, or buckets and each bucket in turn is sorted using a sorting algorithm. It's a stable linear sorting algorithm.

```
BUCKET-SORT(list , n)
1.  Initialize n empty lists/buckets
2. for every list[i]
3.        insert list[i] into bucket[n*list(i)]
4. use a sorting algorithm to sort each bucket
5. combine all buckets after sorting
```

$O(n^2)$ is the worst case while $O(n+k)$ is the average case performance for Bucket Sort. This requires a lot of memory to store those buckets and hence, isn't efficient for large data sets.

4. Recent Work

4.1. Timsort [8]

Timsort is acquired from insertion and merge sort and is hence, a **hybrid stable** algorithm. The algorithm finds already ordered subsequences (run) of the data and uses that in order to sort the remainder. It sorts these runs using the insertion sort and merges these runs using merge sort. It is a standard sorting algorithm in Python, Android JDK and OpenJDK. The algorithm for Timsort is as below.

```
TIMSORT(S,n)
1. R ← run decomposition of S
2. X ← null
3. while R is not null do
4.       R ← pop(R)
5.       Append R to X
6.       while X violates at least one rule do
7.                if |X| <|Z| then
8.                         merge X and Y
9.                else if |X| < = |Y| +|Z| then
10.                        Merge Y and Z
11.               else if |W| <= |X| + |Y| then
12.                        merge Y and Z
13.               else if |Y| <= |Z| then
14.                        merge Y and Z
```

Fig 1. Pseudocode for Timsort

$O(n \log n)$ is its worst case performance where it does $\Theta(n \log n)$ comparisons while Linear when the data is sorted (best case).

4.2. Library Sort[3]

Library Sort or gapped insertion sort makes use of the insertion sort, but with vacancies in the list to speed up the following insertions. This is a stable and a comparison based algorithm.

```
rebalance(A, begin, end)
1.   r ← end
2.   w ← end * 2
3.   while r >= begin
4.      A[w+1] ← gap
5.      A[w] ← A[r]
6.      r ← r - 1
7.      w ← w - 2
```

```
sort(A)
1.   n ← length(A)
2.   S ← new array of n gaps
3.   for i ← 1 to floor(log2(n) + 1)
4.      for j ← 2^i to 2^(i+1)
5.         ins ← binarysearch(A[j],  S, 2^(i-1))
6.         insert A[j] at S[ins]
```

4.3. Introspective sort

This a **hybrid sorting** algorithm with optimal worst case complexity. It begins with quick-sort and continues till the recursion becomes greater than logarithm (number of elements) it moved to heapsort. This is also a comparison sort algorithm.

```
sorting(Arr)
1. max = log(Arr.length) × 2
2. introsort(Arr, max)
```

```
1.introsort(Arr, max)
2.    if Arr.length ≤ 1:
3.        return  // base case
4.    else if max = 0:
5.        heap_sort(Arr)
6.    else:
7.        pivot ← partition(Arr)
8.        introsort(Arr (0…p), max - 1)
9.        introsort(Arr (p+1…n), max - 1)
```

4.4. Spread Sort [6]

This is a hybrid sorting algorithm which combines radix sort and bucket sort (distribution-based sorts) along with that of merge sort and quick sort (comparison based sorting). The worst case performance of this algorithm comes out to be *O(n log n)*. The algorithm uses the partitioning concept of Quick sort and partitions the given list at each step, not into two but into *n/c* segments, where c is a constant and n the count of elements. For these partitions, it locates the smallest and the largest value of the input and partitions the elements between these two into *n/c* segments of same size. By keeping the number of bins as maximum at each step, we can reduce cache misses, and when we have minimum number of bins equal to *n*, this sort converts to bucket sort and completes. Else, the recursive sorting of the each bin is done.

4.5. Block Sort

Block sort is a hybrid in-place stable algorithm which combines insertion sort with merge (at least two) to give a worst case performance of O(n log n) while a best case of O(n). Given two sorted lists, it breaks the first into equal sized blocks and then inserts each of these blocks into the second array and then performs merge of these two.Arne Kutzner and Pok-Son Kim proposed this algorithm in 2008. This a rather lengthy algorithm with several functions involved.

5. Comparison Chart

Algorithm	Worst case	Average Case	Best Case	Space Complexity
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	*O(1)*
Quick Sort	$O(n^2)$	*O(n log n)*	*O(n log n)*	*O(1)*
Merge Sort	*O(n log n)*	*O(n log n)*	*O(n log n)*	*O(1)*
Heap Sort	*O(n log n)*	*O(n log n)*	*O(n log n)*	*O(1)*
Radix Sort	*O(wn)*	-	-	*O(w + N)*
Bucket Sort	$O(n^2)$	*O(n+k)*	$\Omega(n+k)$	*O(n.k)*
Introsort	***O(n log n)***	***O(n log n)***	-	-
Timsort	***O(n log n)***	***O(n log n)***	***O(n)***	***O(n)***
Library Sort	$\mathbf{O(n^2)}$	***O(n log n)***	***O(n)***	***O(n)***

6. References

[1] https://www.infopulse.com/blog/timsort-sorting-algorithm/

[2] https://en.wikipedia.org/wiki/Library_sort

[3] https://en.wikipedia.org/wiki/Hybrid_algorithm

[4] *Data Structures Algorithms Analysis in C++ (Third Edition) by Mark A. Weiss , Prentice Hall.*

[5] *Introduction to Algorithms, 3rd Edition, by T. Cormen, C. Leiserson, R. Rivest, and C. Stein. McGraw-Hill, 2009*

[6] *The Spreadsort High-performance General-case Sorting Algorithm. Parallel and Distributed Processing Techniques and Applications, Steven J. Ross. , Volume 3, pp. 1100–1106. Las Vegas Nevada. 2002.*

[7] *Merge Strategies: from Merge Sort to TimSort by Nicolas Auger, Cyril Nicaud, Carine Pivoteau,2015.*